A Flash File System for Resource Restricted Platforms

Eriko Werbet[1,2], Angelo Brayner[1]

[1] University of Fortaleza, Brazil
brayner@unifor.br
[2] Atlantico Institute, Brazil
eriko@atlantico.com.br

Abstract. A computing device is usually comprised of a processing unit, a volatile memory area (primary memory) and a persistent memory area (secondary memory). Nowadays, in most of mobile computing devices data is persisted in flash memory, which is a kind of non-volatile memory that may be electrically recorded and erased, i.e., an EEPROM (Electrically Erasable Programmable Read-Only Memory) chip. Such memory is also used to implement memory cards, USB sticks and solid state drives. Data persisted in flash memory need to be recorded in a structured way, such that organizing, searching and accessing the data may be done in a transparent way. In other words, efficient file systems should be designed and implemented for flash memory. There are file systems designed for flash memory. Notwithstanding, most of them have not been designed to take into account severe computing resources restrictions, such as low processing power and available primary memory area. In this work, we present a flash file system, denoted Nano-FS, which is quite efficient to run in computing devices with rigid computing restrictions. Nano-FS has been implemented to run in a hardware platform, which was in fact a control board for managing data produced by multiple measurement devices (sensors).

Categories and Subject Descriptors: H.3.0 [**Information Storage and Retrieval**]: General; H.3.2 [**Information Storage and Retrieval**]: Information Storage—*File organization*; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*Retrieval models*

Keywords: embedded system, file system, flash memory, sensor

1. INTRODUCTION

A computing device is usually comprised of a processing unit, a volatile memory area (primary memory) and a persistent memory area (secondary memory). Data in secondary memory is arranged in an abstraction called *file*. The key idea behind the concept of file is to make transparent to users how their data is physically stored in a storage device, which is used as secondary memory. In order to efficiently manage several files, file systems have been proposed [Tanenbaum 2007; Giampolo 1998].

Cell phones, media devices (players, digital cameras, tablets etc.) and the most common mobile computing devices also have a persistent memory area. In the case of most mobile devices, the memory in which the data is persisted is called *flash* [Boboila and Desnoyers 2011; Chen et al. 2009], which is a kind of non-volatile memory that may be electrically recorded and erased, i.e., an EEPROM (Electrically Erasable Programmable Read-Only Memory) chip [Gal and Toledo 2005b]. Such memory is used to implement memory cards, USB sticks and solid state drives, for example.

Flash memory stores data in an array of floating-gate transistors, called cells. Bits are represented by means of the voltage level in a cell. Typically, a flash device is composed of several planes, each of which has a set of blocks. In turn, each block is divided into pages. There are three operations which may be performed on a flash device: read, erase and program [Gal and Toledo 2005b]. A *read* operation may randomly occur anywhere in a flash device. An *erase* operation is applied to a given

*This work was partially funded by Atlantico Institute, Brazil.

block of a flash device and sets all bits to 1. A *program* operation sets a bit to 0. It is important to note that a program operation can only be performed on a "clean" (free) block, which is a block with all bits set to 1.

A NAND flash device is page-addressable for executing read and program operations. Pages are typically of 512, 1024 or 2048 bytes [Hudlet and Schall 2011]. However, the erasing unit in NAND flash memory remains a block (set of pages). Since a program operation can only be applied to clean blocks, in order to change the values of a set of bytes in a block B of a NAND flash device, the entire block should be first erased (i.e., all bits must be set to 1). Thereafter, it is possible to perform the program operation on B, and change the values of the set of bytes. Thus, having a clean block B, any location in B can be programmed, but once a bit in B is set to 0, it is mandatory to erase the entire block to change this one bit to 1. In other words, flash memory offers random-access read and program operations, but does not offer arbitrary random-access rewrite or erase operations.

Naturally, data is stored in flash memory devices by means of file systems. By using an additional software layer over the flash memory [Regan 2009], it is possible to run a traditional file system (FAT, NTFS or ext, for example) over that kind of storage media. That solution is simpler from the implementation perspective, but quite ineffective due to the following reasons:

—**Erasing blocks**: due to the way flash memory works, it is mandatory to erase a physical block before it can be recorded with new data [Gal and Toledo 2005b; Regan 2009]. In addition, the required time to erase blocks may be substantial, depending on how many blocks are supposed to be erased. Traditional file systems have not been designed around this limitation, since erasing blocks in a hard disk is a straightforward operation.

—**Random access**: traditional file systems have been designed for mechanical storage media, where read and write operations rely on the displacement speed of moving parts. Indeed, all operations are optimized to increase the speed of accessing random data, since the cost of a seek operation (a search for random data in the storage media) is too high when its response time is measured. However, there is no lag in seek operations over flash memory, which renders the optimization strategies and design decisions of a traditional file system useless. Indeed, they introduce yet another bottleneck to the system [Gal and Toledo 2005b; Regan 2009].

—**Physical blocks lifetime**: each write operation lessens the flash memory blocks lifetime, hence its lifetime is measured by the amount of operations it may have before it is no longer possible to use it [Gal and Toledo 2005b; Lin et al. 2006; Regan 2009]. However, before the block is isolated from the system, it is necessary to copy its data to another block and update the control structures of the file system, which is done to maintain the integrity of the file in which the isolated block belonged. One strategy to mitigate this effect is to evenly distribute data over the media. Traditional file systems have not been designed to deal with such a limitation [Gal and Toledo 2005b].

Evidently, new file systems need to be designed in order to deal with this new media storage paradigm, mostly due to its limitations. There are file systems designed for flash memory [Wu and Reddy 2011; Jung et al. 2010; Gal and Toledo 2005b; Lin et al. 2006; Regan 2009]. Notwithstanding, most of them have not been designed to take into account extreme computing resources restrictions, such as processing power and available primary memory area. On the hand, several devices which use flash memory as persistent storage are affected by those restrictions.

In this work, we present a flash file system, denoted Nano-FS, which is quite efficient to run in computing devices with rigid computing restrictions. Nano-FS has been implemented to run in a hardware platform, which may be in fact a control board for managing data produced by multiple measurement devices (sensors). In order to read data from the sensors and to process them, a main system is installed in the control board to run continuously. Moreover, the control board has the following restrictions:

—Low processing power microcontroller.

—32 bit architecture.

—64 KB of in-chip volatile memory.

—No operating system, no memory management unit, no dedicated cache area and no multitask support.

The aforementioned main system occupies 56 KB of primary memory. For that reason, the file system itself cannot take more than 8KB of memory to run. Furthermore, there are I/O and processing restrictions. The main system should read data from 8 sensors and write them in secondary memory in cycles of 1s. Thus, the main system has to read, process and write data into flash memory under the 1s mark.

Such an I/O restriction may be hard to enforce, since the control board has no multitask support, and the time to write and erase data is usually an order of magnitude greater than the time to read it in NAND flash memory [Gal and Toledo 2005b; Lin et al. 2006].

Although the system has particular restrictions, one could think of adapting an already tried and proven flash file system, before designing a new file system with all the presented restrictions in mind. However, as will be presented in the next section, the former was not a viable choice for such restrictive scenario.

This article is organized as follows. In the next section, file systems for flash memory devices are described and analyzed. Section 3 presents the proposed file system. In order to evaluate Nano-FS experiments have been carried out, whose results are presented in Section 4. Finally, Section 5 concludes the article.

2. RELATED WORK

Roughly speaking, a flash file system may work as follows. Whenever a given block B has to be updated, a clean block (an unused one) is chosen to receive data from B (the older block) and the data is recorded in it. After that, the control structures are updated (including a data structure that maps physical blocks into logical ones). As one can see, the older block B is not erased right away, since it takes too much time to erase it (this could be a problem if too many blocks are supposed to be erased at once). Thus, B is marked to be physically erased later in the logical block map. At a given time, marked blocks will be physically erased [Gal and Toledo 2005b; Regan 2009].

The behavior of a log-based file system may be used as a model to design a flash file system. This is because the erase algorithm of log-based file systems avoids that the system loses a lot of time when updating files. Moreover, the lifetime of physical blocks in flash devices may be increased, since log-based file systems does not make in-place writes (updates) [Gal and Toledo 2005b; Regan 2009]. Besides, log-based file systems are naturally *journalized*, which means that in the occurrence of system failure or an emergency shutdown, the file system may resume from a point in time just before the failure event [Gal and Toledo 2005b]. One could further extend that by implementing a way to navigate through those versions, and select a version of a file from a week ago, or even use it to recover a file that has been accidentally deleted by the user (like the Time Machine component in Apple's Mac OS X).

There are already flash file systems, which have been implemented based in a log file system model, like the ones presented in this section.

2.1 JFFS2

The JFFS2[1] (Journalling Flash File System version 2) is a log-based file system designed by Red Hat (based on the original JFFS) to be used in embedded systems equipped with flash memory, like cell phones and digital cameras, and its first version is part of the first generation of flash file systems [Gal and Toledo 2005b]. This system has got some noteworthy disadvantages, like the fact that it needs to scan all its nodes (blocks) when the storage media is being mounted by the operating system. This behavior may be critical when used on contemporary devices, which may easily reach 64GB or more, like SSDs (Solid State Drives).

It also implements data compression, which may be a bottleneck in devices with low processing power. Besides, writing small data blocks may lead to negative compression rates (the stored data may be greater than the actual data), forcing applications to use larger write buffers, increasing response time, since the buffer needs to be filled before flushing it to the storage media. That could be even more aggravating in a sensor device [Gal and Toledo 2005a], because it always generates small data blocks from its measurement equipment, and large buffers are out of question, since those devices are already very memory-constrained. In JFFS2 there is no deterministic way to tell how much memory is left available at a given time, since that depends on the writing sequence of the log, and the compression rate of each buffer that needs to be flushed in the future.

In reality, JFFS2 is not suitable for restrictive platforms for the following reasons:

—**Resource usage**: JFFS2 is not a viable choice for devices that need to handle a large amount of data (like a continous stream of sensor data), since its computing resources usage is quite aggressive, mainly due to the way its RAM usage increases in linear proportion to the number of nodes (data units to be stored) [Namihira 2009; Korolev 2007].

—**Overall performance**: JFFS2 performance on applications that mostly run read operations may be acceptable. However, for devices whose applications perform mixed operations or a continuous stream of write operations, it is most likely to be subpar. Besides slow write operations, it requires a large amount of time to mount itself, and it is worsen as the amount of data stored increases [Namihira 2009; Korolev 2007].

—**Slow recovery**: power failure events trigger JFFS2 to check/rebuild the file system data structures from the log, which is a costly operation, specially if the media is already near-capacity, since it requires a large amount of free space to run its recovery procedure[2]. The device is most likely to remain halted for the entire duration of this operation.

—**Inefficient space usage**: JFFS2 will work very inefficiently if the application upon it generates many small writes. Unless and until it is garbage collected, each write operation will take its own JFFS2 node (a block structure) on the media, and so will incur overhead from the node header (the metadata portion of the block). A JFFS2 filesystem is likely to fill up very fast if its data is written a few bytes at a time (like the writes generated by sensors), rather than in large chunks. Small write operations may also remove the benefits of compression entirely, as there is not enough data to compress effectively [Namihira 2009].

—**Operating system dependency**: this filesystem has been designed to run under the Linux kernel, so it depends on it to operate, which makes it harder to adapt to run over platforms with no operating system whatsoever, like embedded systems based in microcontroller architectures [Namihira 2009; Korolev 2007].

[1] http://sources.redhat.com/jffs2/jffs2.pdf
[2] http://www.ecoscentric.com/ecospro/doc/html/ref/fs-jffs2-usage.html

2.2 YAFFS

YAFFS (Yet Another Flash File System) has been designed with JFFS2 in mind (it is also a log-based file system) [Gal and Toledo 2005b; Lin et al. 2006; Regan 2009]. However, it is prepared to deal with storage media larger than 64MB. This system has no built-in data compression, since it assumes that most data formats are already compressed, like music encoded with MP3 and videos encoded by the H.264 codec, for instance.

Although YAFFS has been designed for use with flash memory devices, it is not prepared to deal with devices in the gigabyte range, like solid state drives, and devices used to store lots of static data (like data generated to be analyzed and then stored, like the log of a sensor device). In truth, it is not suitable to run over microcontroller-based devices, mostly due to the following reasons:

—**Wear leveling**: when YAFFS has been designed, the file system itself was supposed to take care of wear leveling (in contrast with wear leveling implemented directly into the hardware controller, like it is implemented nowadays), so it has been implemented with it in mind. However, it only implements dynamic wear leveling, and wear leveling is not performed for static data, which may cause a great number of blocks to be rendered useless at a faster rate than if both static and dynamic wear-leveling algorithms were implemented [Wu and Reddy 2011; Wookey 2007]. This is not suitable for devices that generates lots of static data, since it may greatly decrease the lifespan of the flash media devices attached to it, be it a NAND flash chip or a solid state drive. Since wear leveling is a main feature of this file system, it may impact its performance in devices where wear leveling is performed by the flash controller.

—**Code size**: YAFFS code is around 55 KB in size, so it is not a choice for restricted platforms. That amount would be fine for a device running Linux, but not a choice for a microcontroller-based system with a few kilobytes of RAM. Recall that our embedded system has only 64KB of main memory, and most of it is already used by its main system (see Section1).

—**Run-time data structures**: according to another estimation made by the YAFFS team, it takes around 355 KB (it needs a total amount of 410 KB including its own code) to handle a NAND flash device with 64 MB and 500 files. Moreover, its main data structure, the tnodes make up for 256 KB of the total amount[3].

—**Operating system dependency**: as with JFFS2, this filesystem has also been designed to run under the Linux kernel.

2.3 UBIFS

The Unsorted Block Image File System[4] (UBIFS) has been designed to surpass JFFS2 shortcomings, namely the lack of a cache subsystem for write operations, and a worse-case free memory calculation (JFFS2 uses a best-case free space calculation [Korolev 2007]. It performs better than JFFS2 when handling larger NAND flash devices [Namihira 2009]. Like JFFS2, it does real-time compression of data using the deflate algorithm (zlib) or the LZO (Lempel-Ziv-Oberhumer) algorithm. Unlike JFFS2, UBIFS stores its control data structures in the flash media itself, while the former stores it in main memory, which might take a long time to rebuild, since it needs to read all the file system metadata from flash in order to rebuild its control structures in memory (at mount time). Another relevant difference between JFFS2 and UBIFS is that the latter uses a B_+ tree to index the file system blocks, while JFFS2 uses a linked list.

Although UBIFS might be a better choice for larger NAND flash devices, it is not a viable choice for resource-restricted platforms for the following reasons:

[3]http://www.yaffs.net/node/348
[4]http://www.linux-mtd.infradead.org/doc/ubifs.html

—**Code size**: Since UBIFS depends on the UBI layer of the linux kernel, its size must be included in the code size calculation, which leads to a total of 250 KB [Namihira 2009].

—**RAM comsumption**: According to a benchmark that has been conducted by Toshiba Corp. [Namihira 2009], UBIFS takes almost 600 KB of RAM to handle a file with 1 MB. This may seem reasonable for a system running a operating system, but it is not viable for a device with 1024 KB of main memory, for instance.

—**Metadata overhead**: Also according to the same benchmark by Toshiba [Namihira 2009], UBIFS takes around 10% of the space required to store a single file, to store its own metadata (this could be so due to the indexing structures used by UBIFS). Again, this could be a problem in small flash devices, since a large portion of its space is being used by the file system itself.

2.4 Discussion

The file systems presented in this section have not been designed to handle a scenario in which there are extreme computing resources limitations, a scenario in which the following limitations may apply:

—Low processing power microcontroller.

—Small amount of available RAM, in the kilobytes range.

—No operating system.

—No memory management unit.

—No dedicated cache area.

—No multitask support.

—I/O and synchronization restrictions (like the ones presented in Section 1).

For that reason, we advocate that the proposed file system is quite adequate for applications running on very restrictive computational environments. In addition, existing flash file systems have been designed to solve another class of storage problem. This is more evident if one takes into account that most of them needs a Linux operating system to run, which is not a viable choice for restrictive scenarios.

3. NANO-FS: A LIGHTWEIGHT FLASH FILE SYSTEM

As already mentioned, there are file systems designed for flash memory. Notwithstanding, most of them have not been designed to take into account harsh computing resources restrictions, such as low processing power and available primary memory area. Nano-FS is a lightweight file system designed to run on flash media.

Nano-FS implements four types of blocks: *metadata, file header, bitmap* and *node* (Figure 1(a)). The first type of block is the *metadata block*, which contains the file system metadata: total amount of available storage memory and an authentication key. The authentication key is verified at the file system mount time and has the functionality of identifying whether or not the file system has already been installed. If the key does not match, it indicates that the system has to be installed. Installation consists in writing the first block with the corresponding key and available storage memory, and the mounting process consists in loading that metadata and the file system control structures.

Manipulating a file requires loading a *file header block* and binding it to a *file handle*. A file handle is an in-memory data structure holding a hash representation of the file name and a *file pointer*, which is a single integer that points to the current position (in bytes) of the file being manipulated. The file pointer is needed because a logical file is manipulated as a contiguous list of bytes, in contrast to its physical counterpart, which is a list of scattered blocks of 512 bytes. Hence the need for an in-memory pointer to precisely browse the file data, and mark the last byte that has been modified or read.

Free blocks management is done using a *bitmap block*, which maps used blocks as "1" and free blocks as "0".*File header blocks* contain the file metadata, such as the file name, the file size, the number of node (data) blocks being used, and the address of the first node block which contains the file data. Data belonging to a given file is stored in *node blocks*. Besides file data, each node block stores the following control data: the actual block size and a pointer to the previous node block (depicted in Figure 1(a).

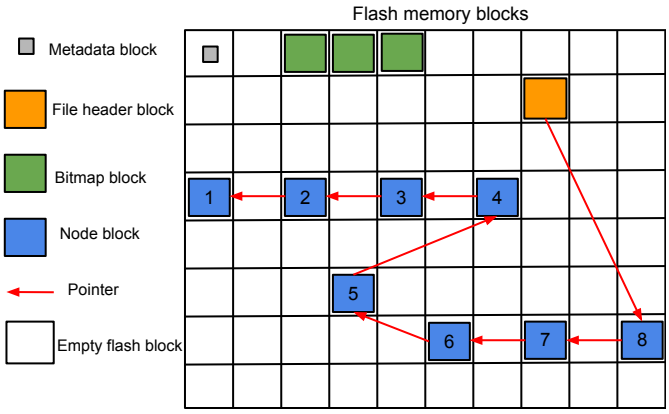
In order to understand how those blocks interact, let us describe a scenario of creating a file of 4096 bytes. A file is created by allocating a file header, which will hold the file metadata. After that, one node block is allocated to hold its data. However, one node block can hold only 500 bytes of file data. A node block is 512 bytes in size, but 12 bytes of it are used for its metadata, like its used size. Thus, more node blocks are allocated, and linked together by means of a list. So far, we have the following block configuration: one file header and several node blocks. Roughly speaking, a file is a list of blocks, where the file header is the head of the list, and it points to the first node holding file data, which points to a second node block, which points to a third node block, and so on and so forth, until all file data is being held by the nodes in the list. The bitmap block role is to keep track of which blocks are being used at a given moment, i.e. free space management. Since a bitmap block (512 bytes) may map up to 2048 blocks (of 512 bytes), we can see that only one bitmap block will suffice. In this case, bitmap block maps itself, the file header and node blocks. Recall that a bitmap block is an array of bits, where a bit set as 0 is a free block, and a bit 1 is an used block. New bitmap blocks may be allocated to keep track of a greater quantity of blocks. Once all blocks have been stored, the file system will update the total amount of space used in the metadata block.

It is important to emphasize that Nano-FS loads only one block of the types bitmap, file header and data node at a given time in order to perform any operation over a single file. This is a very important feature, since it ensures low memory consumption (3 blocks takes 1536 bytes of volatile memory to load).

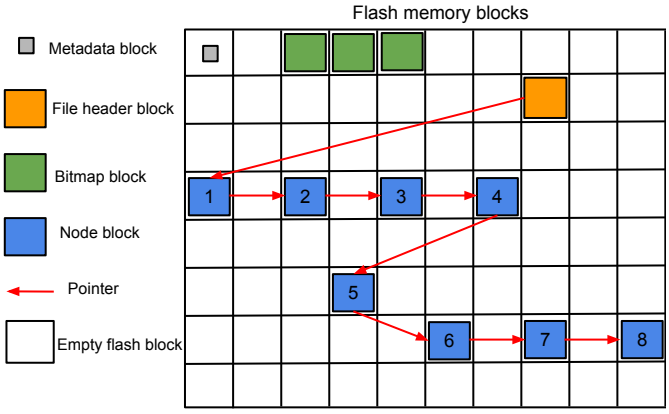
Nano-FS allocates blocks by means of a *reverse linked list*. Files are composed by node blocks (blocks designed to hold file data), which may be scattered along the storage media. This way, those blocks are linked together by a single pointer in each block, which points to the previous block in order to compose a file. A conventional linked list is inefficient in this context, since one would need to write the new block and update the previous block pointer to point to the new one (which is now the tail of the list). Accordingly, such a characteristic would induce the file system to write the flash media twice (three times if one considers that the file header metadata needs to be updated as well) for each newly allocated block. On the other hand, making new blocks point to the last one only requires one write operation, since one just needs to make the new block point to the previous one before the write operation itself. Furthermore, the proposed node-block allocation policy reduces the number of writes in a given block. Consequently, it extends the flash lifetime and decreases the overall time to run write operations.

Next, we describe in details the file-block allocation policy implemented by Nano-FS. For the sake of clarity, let W_a be a stream of write operations for a given $file_a$ (say those operations are related to the creation of the file), where $W_a = (w_1, w_2, w_3, w_4, w_5, w_6, w_7, w_8)$ is the sequence of write operations, and each operation writes a single block of data (each operation fills a single node) into the flash media. Each numbered node i holds the data of write operation $w_i \in W_a$.

Algorithm 1 describes the assembly of a reverse linked list based in the write operations of W_a and Figure 1(a) depicts it. For each write operation w_i the algorithm allocates a new node block N_i. If it is the first node block ($i = 1$), then a new file header must be allocated as well, since it is needed to hold the metadata of the file in question. After that, the file header pointer is updated to point to the new node block (N_1), and the new node block is set to point to *null*, since it currently is the tail of the list. Then the file header is written to the media (line 16 of Algorithm 1), and node N_1 is written as well. When $i > 1$, i.e., from the second block to the last; a new block node $N_{(i>1)}$ is allocated,



(a) Nano-FS reverse linked list.



(b) Conventional linked list allocation.

Fig. 1. Physical data allocation.

then the file header previously written is loaded. The newly allocated node block must point to the previous block $N_{(i-1)}$, in order to keep the blocks linked together. Then the file header metadata is updated, like the file size. The next step is to make the file header point to the new block $N_{(i>1)}$, which is now the first node block in the list(the last written node block becomes the first node block of the list, as seen in Figure 1(a)). After that, the file header is written again, and the new block $N_{(i>1)}$ is written as well (lines 16 and 17 of Algorithm 1).

In order to emphasize the benefits of the proposed file-block allocation policy, consider that the conventional linked list holding the data of $file_a$ would be the one illustrated in Figure 1(b). Recall that each numbered node i holds the data of write operation $w_i \in W_a$. A conventional linked list allocation would be managed by an algorithm such as the one showed in Algorithm 2.

As one can see in Algorithm 2, the last node pointer (the tail of the list) needs to be updated to point to the new node in order to make the new node part of the list. Thus, for each new node to be added into the file list, the allocation algorithm needs to write the flash media three times: updating the file header, updating the tail of the list (previous node) and writing the current block (new tail). Lines 11, 16 and 18 of Algorithm 2. However, having a reverse linked list decreases the amount of write operations needed, as one can see in Algorithm 1.

Comparing Algorithms 2 and 1, one can see that the latter has fewer write operations; two (lines 16 and 17) against three in Algorithm 2 (lines 11, 16 and 18). Having fewer write operations guarantees,

Algorithm 1: Reverse linked list allocation for $file_a$.

```
1  foreach w_i ∈ W_a do
2      Allocate a new node N_i;
3       //i = 1 means its the first node of file_a;
4      if i = 1 then
5          Allocate a new file header with file_a metadata;
6          Make the file header point to N_i;
7          Make N_i point to null;
8      end
9      else
10         Load the file header;
11         Make N_i point to N_(i−1);
12         Update the file size in the file header;
13          //Other file metadata may be updated as well;
14         Update the file header pointer to point to N_i;
15     end
16     Write the file header;
17     Write N_i with data from w_i;
18 end
```

Algorithm 2: Conventional linked list allocation for $file_a$.

```
1  foreach w_i ∈ W_a do
2      Allocate a new node N_i;
3       //i = 1 means its the first node of file_a;
4      if i = 1 then
5          Allocate a new file header with file_a metadata;
6          Make the file header point to N_i;
7      end
8      else
9          Load the previous node N_(i−1);
10         Update the pointer in node N_(i−1) to point to N_i;
11         Write node N_(i−1);
12         Load the file header;
13         Update the file size in the file header;
14          //Other file metadata may be updated as well;
15     end
16     Write the file header;
17     Make N_i point to null;
18     Write N_i with data from w_i;
19 end
```

that a physical data allocation algorithm based in a reverse linked list spends less time writing blocks. In Algorithm 1, only two write operations are performed: updating the file header and writing the current node block. This means, that for each write operation, only two blocks are written into the media. Ergo, it performs better than the one based in a conventional list. The resulting reverse linked list is presented in Figure 1(a).

The algorithm to implement write operations has been designed with the I/O restrictions presented in the previous section in mind. The algorithm first reads the file pointer position, and checks if it is

an append or overwrite operation. In case of an append operation, new node (data) blocks are added to the node list, and the new blocks are marked as used (bit 1) in the corresponding bitmap block, and then the bitmap block is updated.

However, overwrite operations need a different strategy, since overwriting a flash block implies in erasing it and then writing it again, which takes way longer than writing into a "clean" block [Gal and Toledo 2005b; Lin et al. 2006; Regan 2009]. In order to avoid having the write function blocked for a long period of time, we have chosen to always write into clean blocks. Instead of performing an in-place overwrite operation in the block, the data from the old block is updated, and then stored in a clean block. The pointer to each obsolete block is stored in a list, the "dirty" blocks list. Once the media storage starts running out of clean ones, a **garbage collection** routine is triggered.

The garbage collector runs when the system is in idle, reading the dirty blocks list and "cleaning" (erasing) them to be used again by the write algorithm. The collector always try and erase contiguous chunks of data, since it is faster to erase adjacent blocks than individually scattered ones [Choi et al. 2009; Gal and Toledo 2005b; Regan 2009]. Besides, the collector operates in 1s intervals, to synchronize its operation with the possible arrival of new data from the sensors. If the cleaning operation takes more than an estimated duration of 1s, it is rearranged (a different set of dirty blocks is chosen) to clean a feasible quantity of blocks in 1s.

Wear leveling [Wu and Reddy 2011; Jung et al. 2010; Gal and Toledo 2005b] has not been implemented, since the flash circuitry controller already implements that, and bad block management as well.

4. EXPERIMENTAL RESULTS AND ANALYSIS

In order to investigate the performance of Nano-FS, we have implemented a prototype in ANSI C, and performed several experiments, which have been conducted to measure its speed, and the time required to perform a given set of operations. We have not compared it to the file systems in section 2, mostly because those have been designed to run under a Linux operating system, while Nano-FS has been designed to operate without it.

The testbed is comprised of a custom-made control board equipped with a 32-bit microcontroller, 64KB of in-chip volatile memory, and a 4GB NAND flash class 4 SD memory card. Recall that our system runs with no operating system due to memory and processing power constraints (see section 1).

The following parameters have been measured:

—Mount time in miliseconds.
—Throughput in megabytes per second.
—Number of sequential write operations required to write a given file.
—Time required to randomly read and write several files.
—Number of dirty blocks freed by the garbage collector procedure in a given time interval.
—Time required to perform a given set of operations with and without the garbage collector.

Mount time is an important parameter for our scenario, since a fast mount time mostly guarantees a smaller loss of sensor data accuracy due to power oscillations, for example. Likewise, if the system is to be embbeded in a consumer product in the future, it is important to have a fast boot time to avoid having the end-user waiting for it.

This test has been performed by filling the flash memory with several files of 1MB, and then mounting the file system. After that, the media has been completely erased, and the test has been performed again, but with a greater quantity of files, in order to observe how the mounting procedure

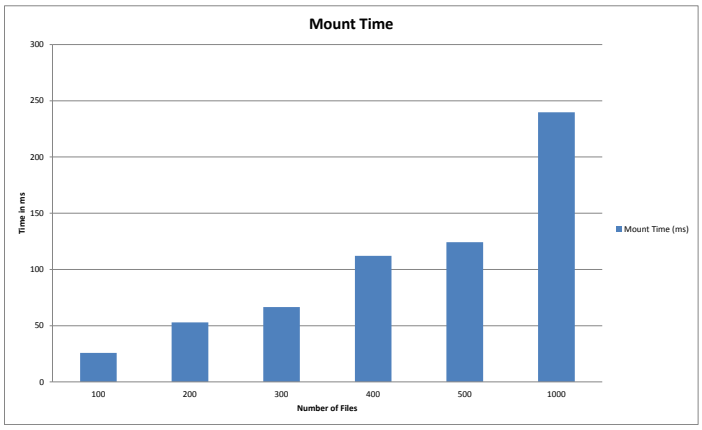


Fig. 2. Mount time versus number of files.

scales. As one can see (in Figure 2), the mount time is barely noticeable for a human being, even with 1000 files being mounted (under 250 ms), since our mounting procedure needs to read only one block of data (the file header) for each file being mounted.

We have also performed throughput tests, in order to observe Nano-FS performance overhead over the NAND flash media nominal throughput. The file system buffer size has also being taken into consideration, since a bigger buffer should translate in greater throughput.

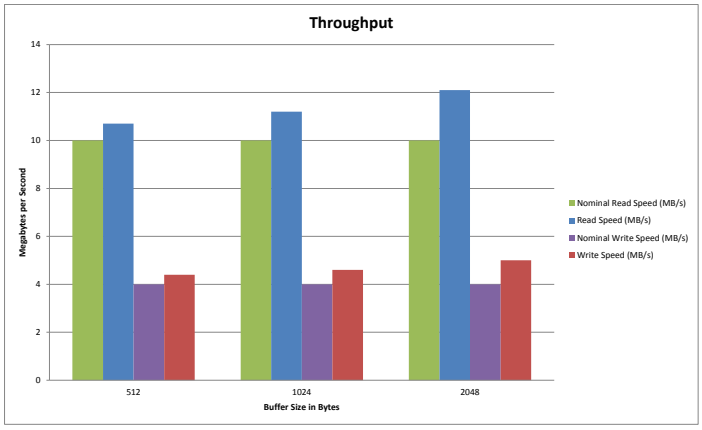


Fig. 3. Throughput versus buffer size.

The throughput tests have been performed by writing and reading a 256 MB file several times, and then computing the average read and write speed for each buffer size. Recall that we are in a resource-constrained platform, so we had to implement small buffers of 512, 1024 and 2048 bytes, and those correspond to one, two and four blocks in size. This means that, instead of loading one block per read operation, we are able to load four blocks using a read buffer of 2048 bytes, and the same applies to write operations as well. As one can see in Figure 3, increasing the buffer size does increase the read speed in a substantial way. However, write operations have been only slightly benefited by the bigger buffer. According to the SD Association[5] (the standards body for the SD card technology), the NAND flash card class 4 has a nominal minimum write speed of 4 MB/s (hence the class 4 designation), and a minimum read speed of 10 MB/s, where the read speed is capped at 12.5 MB/s (spec 1.01), which

[5]https://www.sdcard.org/developers/overview/bus-speed

is the bus interface maximum speed. As a matter of fact, we have achieved 4.4 MB/s write speed with the smaller buffer and 5 MB/s with the biggest one (2048 bytes). Likewise, we have achieved 10.7 MB/s of read speed with the 512 bytes buffer, and 12.1 MB/s with the biggest buffer, which is a relevant mark over the nominal speeds.

Since the write operation takes more time to complete than the read operation, thus becoming the bottleneck in a plataform with a flash media device, one must investigate how the file system's write algorithm scales when handling files of increasing size. In fact, we have measured the number of sequential write operations required to store files of varying sizes.

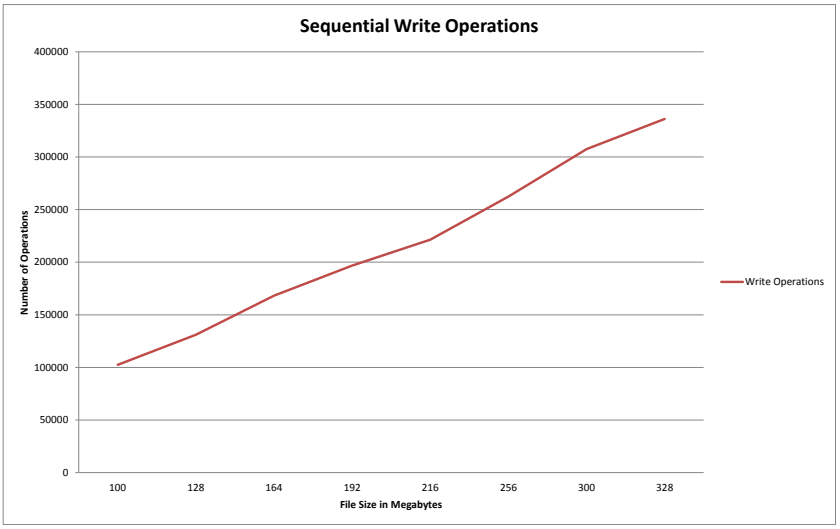


Fig. 4. Number of sequential operations versus file size (linear growth).

The test has been perfomed by writing a single file into the media, then writing another file greater than its predecessor, and so on and so forth, until all files have been written. The chart in Figure 4 shows the number of operations required to store files ranging from 100 MB to 328 MB with a buffer size of 1 KB. In that experiment, the number of operations is directly proportional to the file size (as expected), thus increasing the file size in a linear way, also increases the number of operations linearly. Another set of tests have been performed, in which we increased the file size non-linearly. As one can see in Figure 5, raising the file size in a exponential fashion (2^x), also increases the number of operations exponentially, which was the behavior we were expecting.

We have also observed the file system behavior when dealing with random read and write operations. In this test, we were interested in observe its responsiveness in a scenario where read and write operations could randomly occur, like when reading and writing data belonging to different files. In fact, the time required to operate over a stream of read and write operations has been measured. In this experiment (see Figure 6), we have randomly sorted the read and write operations from different streams (produced by two different files in the megabytes range), where the order of a given stream is kept intact, but both streams are intertwined.

For the sake of clarity, let R_a be a stream of read operations for a given $file_a$, and W_b a write stream for a given $file_b$, where $R_a = (r_1, r_2, r_3, r_4, r_5)$, and $W_b = (w_1, w_2, w_3)$. A valid randomly intertwined stream would be $I_1 = (r_1, r_2, w_1, r_3, r_4, w_3, r_5)$ or $I_2 = (r_1, w_1, r_2, w_2, w_3, r_3, r_4, r_5)$ for instance, or any combination where the order of precedence of the read and write operations is kept for both streams, in order to guarantee that the intended effect of each stream (say read the first 1KB of $file_a$, and append 2KB to $file_b$) is achieved for both files. We have measured the time required to complete each set of read and write operations, while varying the buffer size, as seen in Figure 6. As expected,

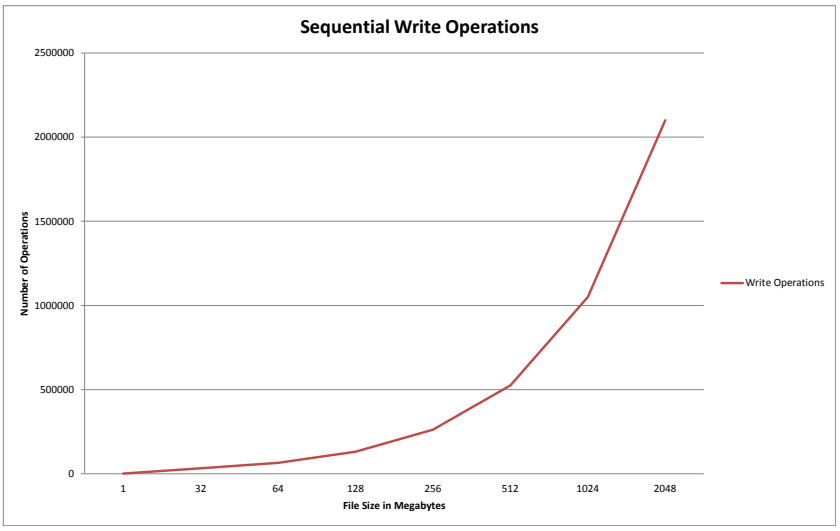


Fig. 5. Number of sequential operations versus file size (exponential growth).

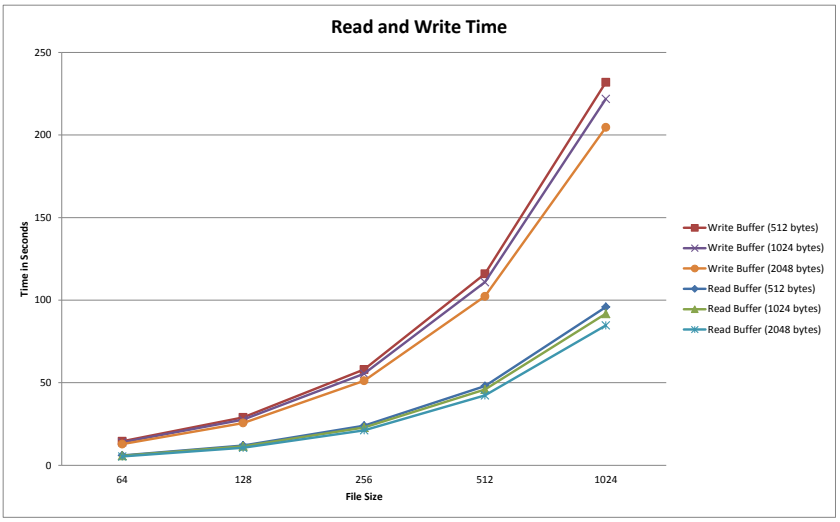


Fig. 6. Random read and write operations.

the time to write a given file is way greater than the time required to read it. However, increasing the buffer size has proved to be advantageous, since it decreased the overall time to read and write a given file (recall Figure 3). It is important to mention, that having a reverse linked list as the physical allocation structure has sped up the write streams, since it only takes two write operations to store each file block, instead of the three operations needed by a conventional linked list of blocks (recall Section 3).

All the aforementioned tests have been conducted without the need to trigger the garbage collector, since the media has been completely erased between tests. Thus, we have performed another set of experiments to measure its efficiency. Recall that the garbage collector is triggered once the media has run out of fresh (clean) blocks, say at 75% of its capacity (3 GB). In fact, we have filled the media until the 75% threshold has been reached, then we marked all those blocks as "dirty", i.e., they have been marked as targets to the collector. After that, we started writing several new files to the media in order to trigger the garbage collection routine. Additionally, we have set increasingly idle time

intervals between each stream of write operations, in order to observe how the collector scales when it has more time to run.

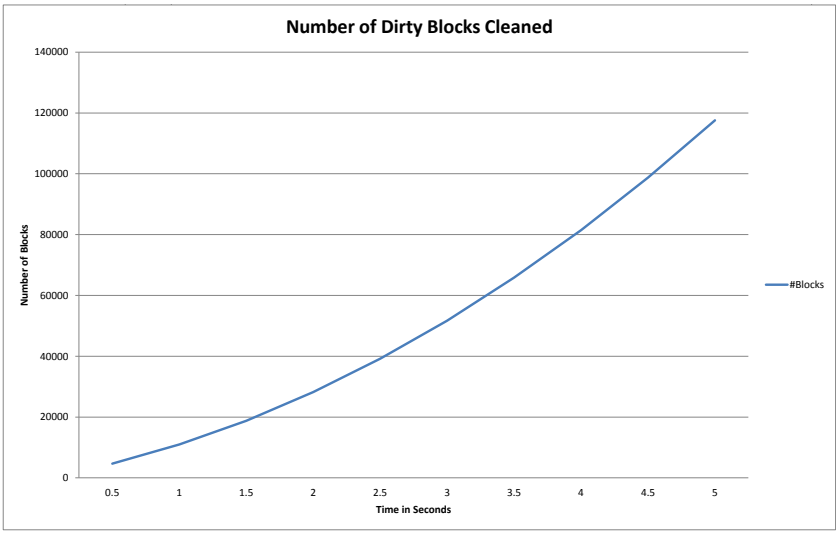


Fig. 7. Number of obsolete blocks erased versus the time.

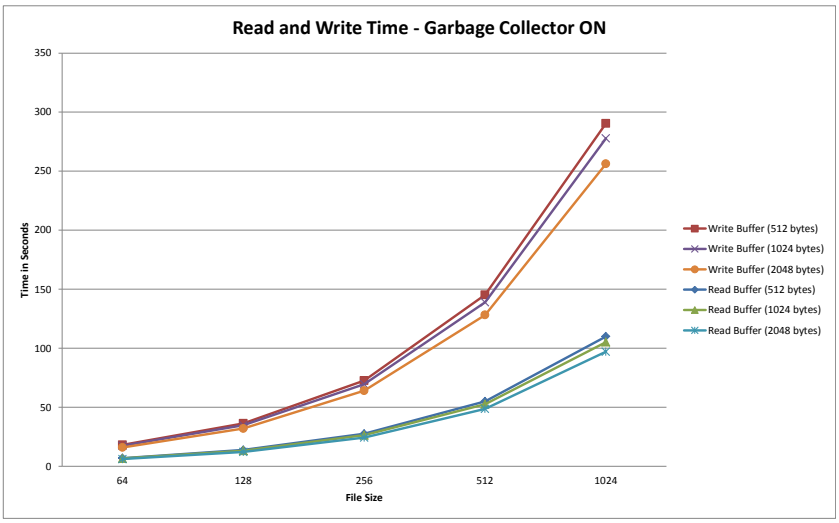
As one can see in Figure 7, as we increase the time in which the collector is allowed to run, the steeper is the rise of the curve that displays the quantity of blocks that have been erased. This exponential efficiency is due to the fact that having more time, makes the collector erase greater chunks of adjacent dirty blocks, in contrast to choosing scattered blocks, which is slower [Gal and Toledo 2005b; Regan 2009].

In the experiment displayed in Figure 6 we have not activated the garbage collection, since there were no update operations involved. However, we have used that experiment as a template, changing it in a way that the garbage collector could be trigged. To achieve that, we have first filled the media up to 90% of its capacity then marked all blocks as dirty, and proceeded with the random operations portrayed in Figure 6. We have also set a fixed idle time interval of 1s every 5s, in order to provide the time needed by the collector to operate.

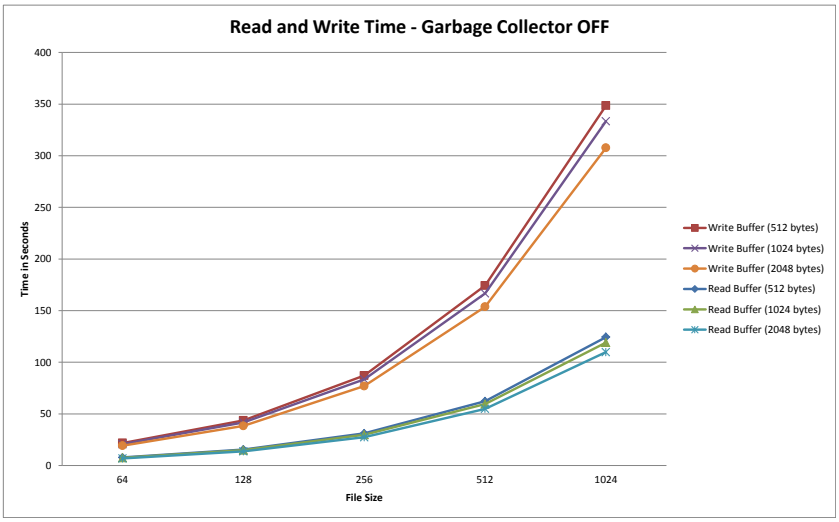
Comparing the experiments of Figure 6 and Figure 8(a)(sizes in megabytes), one is able to see that the required time to perform the random set of operations from Figure 6 has substantially risen. That behavior occurs mostly due to the fact that we have now a massive quantity of dirty blocks, which need to be erased before they can be used again. In fact, one can see that the performance impact is greater in the write operations, since we have added the time required to erase the dirty blocks.

Although the time required to perform the random set of operations has increased, it could be way worse without the garbage collector, so we have turned it off and have performed the test in Figure 8(a) again, in order to gather evidence to back up that assumption. As expected, the overall performance of Nano-FS has been greatly impacted as one can see in Figure 8(b).

Besides being efficient (as seen in the presented charts), Nano-FS is light on RAM usage as well, since it needs to load only the following data to operate: a file header (512 bytes); 64 file handles (512 bytes); a bitmap block (512 bytes); a garbage collector block (512 bytes); a node block (512 bytes), and; an internal buffer area (512 to 2048 bytes). Furthermore, Nano-FS's code itself is around 3 KB in size. As such, the file system takes as low as 6 KB ($6 \cdot 512$ bytes + 3 KB from its code) to run, and a maximum of 8 KB (with a buffer area of 2 KB).



(a) Garbage collector on.



(b) Garbage collector off.

Fig. 8. Time to randomly read and write different file sizes.

5. CONCLUSIONS

We have designed and implemented a flash file system denoted Nano-FS, which is the subject of this work. Nano-FS has been designed to be efficient and light on resources usage. As presented in the Experimental Results section, we have achieved those design objectives, since the file system is quite efficient in storing and retrieving data from a flash memory device.

As a matter of fact, Nano-FS is a viable choice for resource restricted platforms due to the following reasons:

—**RAM usage footprint:** in order to operate, Nano-FS needs to load only 3 KB worth of data, and its code is around 3 KB as well.

—**Efficient physical data allocation:** its data allocation algorithm (Section 3) has been designed to decrease the number of write operations. Thus, reducing the time in which the write function

remains blocked (write operations takes more time than read operations in flash memory).

—**Log-based operation:** as presented, Nano-FS write algorithm always choose to write into free blocks, preventing in-place update operations, whose cost in time is too high.

—**Garbage collector:** since its write algorithm always choose to write into free blocks, it is paramount to have a fast garbage collector to clean those blocks at an opportune time. The implemented collector efficiency scales with the time available to clean adjacent blocks, as presented in Section 4.

—**Fast mount operation:** its mount time is barely noticeable, even when having a thousand files in the flash media.

—**High throughput:** its lightweight design and efficiency enables it to have a high throughput, and it might have room for performance gains with an increase in its buffer size (recall Section 4).

Although Nano-FS has been designed to solve a specific problem, it has been implemented to be easily modified and extended, in order to be used in another devices and scenarios. If the target platform has more volatile memory, one can easily change its behavior to load more data blocks at once (increasing its internal buffer), for instance. One may also change the way the garbage collector operates, replacing its "idle behavior" by a more aggressive behavior, for example.

Future work in this subject may include modifications and extensions to Nano-FS, in order to enable it to run under the Linux operating system, for instance.

REFERENCES

Boboila, S. and Desnoyers, P. Performance Models of Flash-based Solid-state Drives for Real Workloads. In *Proceedings of the 2011 IEEE 27th Symposium on Mass Storage Systems and Technologies*. MSST '11. IEEE Computer Society, Washington, DC, USA, pp. 1–6, 2011.

Chen, F., Koufaty, D. A., and Zhang, X. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems*. SIGMETRICS '09. ACM, New York, NY, USA, pp. 181–192, 2009.

Choi, H. J., Lim, S.-H., and Park, K. H. FTL: A Flash Translation Layer Based on a Journal Remapping for Flash Memory. *ACM Transactions on Storage (TOS)* 4 (4): 4:1–4:22, 2009.

Gal, E. and Toledo, S. A Transactional Flash File System for Microcontrollers. In *Proceedings of the 2007 USENIX Technical Conference*. Advanced Computing Systems Association - USENIX, Santa Clara, CA, USA, pp. 7–7, 2005a.

Gal, E. and Toledo, S. Algorithms and Data Structures for Flash Memories. *ACM Computing Surveys* 37 (2): 138–163, 2005b.

Giampolo, D. *Practical File System Design*. Morgan Kaufmann, 1998.

Hudlet, V. and Schall, D. SSD ≠ SSD - An Empirical Study to Identify Common Properties and Type-specific Behavior. In *14. GI-Fachtagung Datenbanksysteme für Business, Technologie und Web (BTW 2011), Kaiserslautern, Germany*. LNI, vol. P - 180. pp. 430–441, 2011.

Jung, J., Won, Y., Kim, E., Shin, H., and Jeon, B. FRASH: Exploiting Storage Class Memory in Hybrid File System for Hierarchical Storage. *ACM Transactions on Storage (TOS)* 6 (1): 1–25, 2010.

Korolev, A. Improving JFFS2 RAM Usage and Performance. In *Proceedings of CE Linux Forum Embedded Linux Conference*. CE Linux Forum, CE Linux Forum, Linz, Austria, pp. 9–9, 2007.

Lin, S., Zeinalipour-Yazti, D., Kalogeraki, V., Gunopulos, D., and Najjar, W. Efficient Indexing Data Structures for Flash-based Sensor Devices. *ACM Transactions on Storage (TOS)* 2 (4): 468–503, 2006.

Namihira, S. Evaluation of Flash File Systems for Large NAND Flash Memory. In *Proceedings of CE Linux Forum Embedded Linux Conference*. CE Linux Forum, CE Linux Forum, Grenoble, France, pp. 9–9, 2009.

Regan, J. *The forensic potential of flash memory*. Ph.D. thesis, Naval Postgraduate Scholl, Monterey, CA, USA, 2009.

Tanenbaum, A. *Modern Operating Systems*. Prentice Hall, 2007.

Wookey, . YAFFS - A NAND Flash File System. In *Proceedings of CE Linux Forum Embedded Linux Conference*. CE Linux Forum, CE Linux Forum, Linz, Austria, pp. 7–7, 2007.

Wu, X. and Reddy, A. L. N. SCMFS: A File System for Storage Class Memory. In *Proceedings of the 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, Seattle, WA, USA, 2011.